

Liquid Intersection Types

Mário Pereira Sandra Alves Mário Florido

University of Porto, Department of Computer Science & LIACC

{mariopereira,sandra,amf}@dcc.fc.up.pt

We present a new type system combining refinement types and the expressiveness of intersection type discipline. The use of such features makes it possible to derive more precise types than in the original refinement system. We have been able to prove several interesting properties for our system (including subject reduction) and developed an inference algorithm, which we proved to be sound.

1 Introduction

Refinement types [9] state complex program invariants, by augmenting type systems with logical predicates. A refinement type of the form $\{v : B \mid \phi\}$ stands for the set of values from basic type B restricted to the filtering predicate (refinement) ϕ . A subtyping relation exists for refinement types, which will generate implication conditions:

$$\frac{\Gamma; v : B \vdash \phi \Rightarrow \psi}{\Gamma \vdash \{v : B \mid \phi\} <: \{v : B \mid \psi\}}$$

One idea behind the use of such type systems is to perform type-checking using SMTs (Satisfiability Modulo Theories) solvers [14], discharging conditions as the above $\phi \Rightarrow \psi$. However, the use of arbitrary boolean terms as refinement expressions leads to undecidable type systems, both for type checking and inference.

Liquid Types [13, 15] present a system capable of automatically infer refinement types, by means of two main restrictions to a general refinement type system: refinement predicates of some terms are conjunctions of expressions exclusively taken from a global, user-supplied set (denoted $\mathbb{Q}$) of logical qualifiers (simple predicates over program variables, the value variable v and the variable placeholder $\star$); and a conservative (hence decidable) notion of subtyping.

Despite the interest of Liquid Types, some situations arise where the inference procedure infers poorly accurate types. For example, considering $\mathbb{Q} = \{v \geq 0, v \leq 0\}$ and the term $neg \equiv \lambda x. -x$, Liquid Types infer for neg the type $x : \{0 \leq v \wedge 0 \geq v\} \rightarrow \{0 \leq v \wedge 0 \geq v\}$ (throughout this paper we write $\{\phi\}$ instead of $\{v : B \mid \phi\}$ whenever B is clear from the context). This type cannot be taken as a precise description of the neg function's behavior, since it is not expressed that for a positive (resp. negative) argument the function returns a negative (resp. positive) value. With our system we will have for neg the type $(x : \{v \geq 0\} \rightarrow \{v \leq 0\}) \cap (x : \{v \leq 0\} \rightarrow \{v \geq 0\})$.

We introduce *Liquid Intersection Types*, a refinement type system with the addition of intersection types [2, 3]. Our use of intersection in conjunction with refinement types is motivated by a problem clearly identified for Liquid Types: the absence of most-general types, as in the ML tradition. Our use of intersection for refinement types draws some inspiration from [6], since this offers a way to use jointly detailed types and intersections. Though, integrating this expressiveness with refinement types and keeping the qualifiers from $\mathbb{Q}$ simple (which must be provided by the programmer) implies the design of a new type system.

M, N	$::=$	x c $\lambda x. M$ MN $\text{let } x = M \text{ in } N$ $[\Lambda\alpha]M$ $[\tau]M$	<i>Terms:</i> variable constant abstraction application let-binding type abstraction type instantiation
ϕ	$::=$	q $\top$	<i>Liquid refinements:</i> qualifier from $\mathbb{Q}$ true (empty refinement)
B	$::=$	int bool	<i>Base types :</i> integers booleans
$\tilde{\tau}(\mathcal{R})$	$::=$	$\{v : B \mid \mathcal{R}\}$ $x : \tilde{\tau}(\mathcal{R}) \rightarrow \tilde{\tau}(\mathcal{R})$ $\tilde{\tau}(\mathcal{R}) \cap \tilde{\tau}(\mathcal{R})$ α	<i>Pretype skeleton</i> base refined type function intersection type variable
$\tilde{\sigma}(\mathcal{R})$	$::=$	$\tilde{\tau}(\mathcal{R})$ $\forall \alpha. \tilde{\sigma}(\mathcal{R})$	<i>Pretype scheme skeleton :</i> mono pretype pretype scheme
T	$::=$	B α $T_1 \rightarrow T_2$	<i>Simple types:</i> basic type type variable functional type
$\dot{\tau}(\mathcal{R}), \dot{\sigma}(\mathcal{R})$	$::=$	$\tilde{\tau}(\mathcal{R}) :: T, \tilde{\sigma}(\mathcal{R}) :: T$	<i>Well-founded pretype, scheme</i>
τ, σ	$::=$	$\dot{\tau}(E), \dot{\sigma}(E)$	<i>Refinement Intersection Type, Scheme</i>
$\hat{\tau}, \hat{\sigma}$	$::=$	$\dot{\tau}(\phi), \dot{\sigma}(\phi)$	<i>Liquid Intersection Type, Scheme</i>
Γ	$::=$	$\emptyset$ $\Gamma; x : \sigma$	<i>Environment:</i> empty new binding

Figure 1: Syntax

Besides the new type system, another contribution of this work is a new inference algorithm for Liquid Intersection Types.

This paper is organized as follows. Section 2 presents the designed type system, with a focus on the language syntax, semantics and typing rules, as well as a soundness result. The type inference algorithm is introduced in section 3. Finally, in section 4 we conclude with final remarks and explain some possible future work.

2 Type system

2.1 Syntax and semantics

Our target language is the λ -calculus extended with constants and, as in the Damas-Milner type system, local bindings via the `let` constructor. We assume the Barendregt convention regarding names of free

and bound variables [1], and identify terms modulo α -equivalence. The syntax of expressions and types is presented in Figure 1.

The set of constants of our language is a countable alphabet of constants c , including literals and primitive functions. We assume for primitive functions the existence of at least arithmetic operators, a fixpoint combinator `fix` and an identifier representing `if-then-else` expressions. The type of constants is established using a mapping $ty(c)$, assigning a refined type that captures the semantic of each constant. For instance, to an integer literal n it would be assigned the type $\{v : int \mid v = n\}$. Note that refinements may come from the user defined set $\mathbb{Q}$ or from the constants and sub-derivations. In the latter case the refinement expressions are arbitrary expressions from E .

We use $\tilde{\tau}(\mathcal{R})$ and $\tilde{\sigma}(\mathcal{R})$ to denote pretypes and pretype schemes, respectively (this notion of pretypes goes back to [11]), which stand for type variables, basic and functional refined types, intersection of pretypes and polymorphic pretypes. The notation $x : \tau_1 \rightarrow \tau_2$ will be preferred over the usual $\Pi(x : \tau_1). \tau_2$ for functional dependent types, meaning that variable x may occur in the refinement expressions present in τ_2 . An intersection in pretypes (denoted by $\cap$) indicates that a term with type $\tilde{\tau}_1(\mathcal{R}) \cap \tilde{\tau}_2(\mathcal{R})$ has both type $\tilde{\tau}_1(\mathcal{R})$ and $\tilde{\tau}_2(\mathcal{R})$, respecting the possible refinement predicates figuring in these types. We assume the ' $\cap$ ' operator to be commutative, associative and idempotent.

A *well-founded pretype* (resp. *well-founded type scheme*) is a pretype $\tilde{\tau}(\mathcal{R})$ (resp. $\tilde{\sigma}(\mathcal{R})$) for such that $\tilde{\tau}(\mathcal{R}) :: T$ (resp. $\tilde{\sigma}(\mathcal{R}) :: T$), for some T (T stands for simple types for the rest of this document). The *well-founded* relation $\tilde{\sigma}(\mathcal{R}) :: T$ is inductively defined by:

$$\begin{array}{c} \text{::-FUN} \qquad \qquad \qquad \text{::-REF} \qquad \qquad \qquad \text{::-}\forall \qquad \qquad \qquad \text{::-}\cap \\ \frac{\alpha :: \alpha}{\alpha :: \alpha} \quad \frac{\tilde{\tau}_x(\mathcal{R}) :: T_x \quad \tilde{\tau}(\mathcal{R}) :: T}{(x : \tilde{\tau}_x(\mathcal{R}) \rightarrow \tilde{\tau}(\mathcal{R})) :: T_x \rightarrow T} \quad \frac{\{v : B \mid \mathcal{R}\} :: B}{\{v : B \mid \mathcal{R}\} :: B} \quad \frac{\tilde{\sigma}(\mathcal{R}) :: T}{\forall \alpha. \tilde{\sigma}(\mathcal{R}) :: T} \quad \frac{\tilde{\tau}_1(\mathcal{R}) :: T \quad \tilde{\tau}_2(\mathcal{R}) :: T}{\tilde{\tau}_1(\mathcal{R}) \cap \tilde{\tau}_2(\mathcal{R}) :: T} \end{array}$$

Using this relation guarantees that intersection of types are at the refinement expressions only, i.e. for $\tilde{\sigma}_1(\mathcal{R}) \cap \tilde{\sigma}_2(\mathcal{R})$ both $\tilde{\sigma}_1(\mathcal{R})$ and $\tilde{\sigma}_2(\mathcal{R})$ are of the same form, solely differing in the refinement predicates.

To describe the execution behavior of our language we use a small-step contextual operational semantics, whose rules are shown in Figure 2. The relation $M \rightsquigarrow N$ describes a single evaluation step from term M to N . The rules $[\mathcal{E} - \beta]$, $[\mathcal{E} - Let]$ and $[\mathcal{E} - Compat]$ are standard for a call-by-value ML-like language. The rule $[\mathcal{E} - Constant]$ evaluates an application with a constant in the function position. This rule relies on the embedding $\llbracket \cdot \rrbracket$ of terms into a decidable logic [10] (the definition of this embedding, as well as the details of the used logic, will be made clear in next section).

2.2 Typing rules

We present our typing rules via the collection of derivation rules shown in Figure 3. We present three different judgments: **type judgment**, of the form $\Gamma \vdash_{\mathbb{Q}} M : \sigma$ meaning that term M has type σ under environment Γ , restricted to the qualifiers contained in $\mathbb{Q}$ i.e., only expressions from the set $\mathbb{Q}$ can be used as refinement predicates for the following terms: let bindings, λ -abstractions and type instantiations; **subtype judgment** $\Gamma \vdash^{\cap} \sigma_1 \prec \sigma_2$, stating that σ_1 is a subtype of σ_2 under the conditions of environment Γ ; and the **well-formedness judgment** $\Gamma \vdash^{\cap} \sigma$ indicating that variables referred by the refinements of σ are in the scope of corresponding expressions. The well-formedness judgment can be lifted to well-formedness of environments, by stating that an environment is well-formed if for every binding types are well-formed with respect to the prefix environment. This well-formedness restriction implies the

$V ::=$	c	Values:
	$\lambda x.M$	constant
Contexts		abstraction
$C ::=$	$[\]$	$\boxed{C}$
	CM	Contexts :
	VC	hole
	$\text{let } x = C \text{ in } M$	left application
Evaluation		right application
		let-context
		$\boxed{M \rightsquigarrow N}$
$cV \rightsquigarrow$	$\llbracket c \rrbracket (V)$	$[\mathcal{E} - \text{Constant}]$
$(\lambda x.M)V \rightsquigarrow$	$[V/x]M$	$[\mathcal{E} - \beta]$
$\text{let } x = V \text{ in } M \rightsquigarrow$	$[V/x]M$	$[\mathcal{E} - \text{Let}]$
$C[M] \rightsquigarrow$	$C[N] \text{ if } M \rightsquigarrow N$	$[\mathcal{E} - \text{Compat}]$

Figure 2: Small-step operational semantics.

absence of the structural property of exchange in our system, since by permuting the bindings in Γ one could generate an inconsistent environment

The rule [APP] conforms to the dependent types discipline, since the type of an application MN is the return type of M but with every occurrence of x in the refinements substituted by N .

Another point worth mentioning is the distinction made when the type of a variable is to be retrieved, rules [VAR-B] and [VAR]. Whenever the type of the variable z is an intersection of refined basic type we ignore these refinements and assign z the type $\{v : B \mid v = z\}$, for some basic type B . This is inspired on the system of Liquid types [13], since this assigned refined type is very useful when it comes to use in subtyping, especially with the rule [$\prec$ -Base]. When this is not the case, the type of a variable is the one stored in Γ .

One novel aspect of this system is the presence of the [INTERSECT] rule, which allows to intersect two types that have been derived for the same term. The use of this rule increases the expressiveness of the types language itself, since more detailed types can be derived for a program.

The subtyping relation presents some typical rules for a system with intersection types. These allow to capture the relations at the level of intersections in types, with no concern for the refinements of the two types being compared. On the other side, comparing two refined base types reduces to the check of an implication formula between the refinement expressions. Our system uses a decidable notion of implication in the rule [$\prec$ -Base], by embedding environments and refinement expressions into a decidable logic. This logic contains at least equality, uninterpreted functions and linear arithmetic. This is the core logical setting of most state-of-the-art SMT solvers. The embedding $\llbracket M \rrbracket$ translates the term M to the correspondent one in the logic (if it is the case M is a constant or an arithmetic operator), or if M is a λ -abstraction or an application encodes it via uninterpreted functions. The embedding of environments is defined as

$$\llbracket \Gamma \rrbracket \triangleq \bigwedge \{ (\llbracket E_1 \rrbracket \wedge \dots \wedge \llbracket E_n \rrbracket)[x/v] \mid x : \{v : B \mid E_1\} \cap \dots \cap \{v : B \mid E_n\} \in \Gamma \}$$

Given that every implication expression generated in rule [$\prec$ -Base] is decidable, it is then suitable to be discharged by some automatic theorem prover. So, type-checking in our system can be seen as a *typing-and-proof* process.

We show an example of a derivation for the term $\lambda x. -x$, assuming $\mathbb{Q} = \{v \geq 0, v \leq 0\}$. With

Liquid Intersection Type checking

$$\boxed{\Gamma \vdash_{\mathbb{Q}} M : \sigma}$$

$$\begin{array}{c}
\text{SUB} \\
\frac{\Gamma \vdash_{\mathbb{Q}} M : \sigma_1 \quad \Gamma \vdash^{\cap} \sigma_1 \prec \sigma_2 \quad \Gamma \vdash^{\cap} \sigma_2}{\Gamma \vdash_{\mathbb{Q}} M : \sigma}
\end{array}
\quad
\begin{array}{c}
\text{INTERSECT} \\
\frac{\Gamma \vdash_{\mathbb{Q}} M : \tau_1 \quad \Gamma \vdash_{\mathbb{Q}} M : \tau_2 \quad \tau_1 \cap \tau_2 :: T}{\Gamma \vdash_{\mathbb{Q}} M : \tau_1 \cap \tau_2}
\end{array}$$

$$\begin{array}{c}
\text{VAR-B} \\
\frac{\Gamma(x) = \tau_1 \cap \dots \cap \tau_n \quad \tau_i :: B(\forall i : 1 \leq i \leq n)}{\Gamma \vdash_{\mathbb{Q}} x : \{v : B \mid v = x\}}
\end{array}
\quad
\begin{array}{c}
\text{VAR} \\
\frac{\Gamma(x) \text{ not a base type} \quad \Gamma(x) :: T}{\Gamma \vdash_{\mathbb{Q}} x : \Gamma(x)}
\end{array}$$

$$\begin{array}{c}
\text{APP} \\
\frac{\Gamma \vdash_{\mathbb{Q}} M : (x : \tau_n \rightarrow \tau) \quad \Gamma \vdash_{\mathbb{Q}} N : \tau_x}{\Gamma \vdash_{\mathbb{Q}} MN : [N/x]\tau}
\end{array}
\quad
\begin{array}{c}
\text{FUN} \\
\frac{\Gamma; x : \hat{\tau}_x \vdash_{\mathbb{Q}} M : \hat{\tau} \quad \Gamma \vdash^{\cap} \hat{\tau} \quad \hat{\tau} :: T}{\Gamma \vdash_{\mathbb{Q}} \lambda x. M : (x : \hat{\tau}_x \rightarrow \hat{\tau})}
\end{array}$$

$$\begin{array}{c}
\text{CONST} \\
\frac{}{\Gamma \vdash_{\mathbb{Q}} c : \text{ty}(c)}
\end{array}
\quad
\begin{array}{c}
\text{LET} \\
\frac{\Gamma \vdash_{\mathbb{Q}} M : \sigma \quad \Gamma; x : \sigma \vdash_{\mathbb{Q}} N : \hat{\tau} \quad \Gamma \vdash^{\cap} \hat{\tau}}{\Gamma \vdash_{\mathbb{Q}} \text{let } x = M \text{ in } N : \hat{\tau}}
\end{array}$$

$$\begin{array}{c}
\text{GEN} \\
\frac{\Gamma \vdash_{\mathbb{Q}} M : \sigma \quad \alpha \notin \Gamma}{\Gamma \vdash_{\mathbb{Q}} [\Lambda \alpha] M : \forall \alpha. \sigma}
\end{array}
\quad
\begin{array}{c}
\text{INST} \\
\frac{\Gamma \vdash_{\mathbb{Q}} M : \forall \alpha. \sigma \quad \Gamma \vdash^{\cap} \hat{\tau} \quad \text{Shape}(\hat{\tau}) = T}{\Gamma \vdash_{\mathbb{Q}} [T] M : [\hat{\tau}/\alpha] \sigma}
\end{array}$$

Subtyping

$$\boxed{\Gamma \vdash^{\cap} \sigma_1 \prec \sigma_2}$$

$$\begin{array}{c}
\prec\text{-BASE} \\
\frac{\text{Valid}(\llbracket \Gamma \rrbracket \wedge (\llbracket E_1 \rrbracket \wedge \dots \wedge \llbracket E_n \rrbracket) \Rightarrow (\llbracket E'_1 \rrbracket \wedge \dots \wedge \llbracket E'_m \rrbracket))}{\Gamma \vdash^{\cap} \{v : B \mid E_1\} \cap \dots \cap \{v : B \mid E_n\} \prec \{v : B \mid E'_1\} \cap \dots \cap \{v : B \mid E'_m\}}
\end{array}$$

$$\begin{array}{c}
\prec\text{-INTERSECT-FUN} \\
\frac{}{\Gamma \vdash^{\cap} (x : \tau_x \rightarrow \tau_1) \cap (x : \tau_x \rightarrow \tau_2) \prec (x : \tau_x \rightarrow \tau_1 \cap \tau_2)}
\end{array}
\quad
\begin{array}{c}
\prec\text{-ELIM} \\
\frac{}{\Gamma \vdash^{\cap} \tau_1 \cap \tau_2 \prec \tau_i} \quad i \in \{1, 2\}
\end{array}$$

$$\begin{array}{c}
\prec\text{-FUN} \\
\frac{\Gamma \vdash^{\cap} \tau'_x \prec \tau_x \quad \Gamma; x : \tau'_x \vdash^{\cap} \tau \prec \tau'}{\Gamma \vdash^{\cap} x : \tau_x \rightarrow \tau \prec x : \tau'_x \rightarrow \tau'}
\end{array}
\quad
\begin{array}{c}
\prec\text{-VAR} \\
\frac{}{\Gamma \vdash^{\cap} \alpha \prec \alpha}
\end{array}$$

$$\begin{array}{c}
\prec\text{-INTERSECT} \\
\frac{\Gamma \vdash^{\cap} \tau \prec \tau_1 \quad \Gamma \vdash^{\cap} \tau \prec \tau_2}{\Gamma \vdash^{\cap} \tau \prec \tau_1 \cap \tau_2}
\end{array}
\quad
\begin{array}{c}
\prec\text{-POLY} \\
\frac{\Gamma \vdash^{\cap} \sigma_1 \prec \sigma_2}{\Gamma \vdash^{\cap} \forall \alpha. \sigma_1 \prec \forall \alpha. \sigma_2}
\end{array}$$

Well formed types

$$\boxed{\Gamma \vdash^{\cap} \sigma}$$

$$\begin{array}{c}
\text{WF-B} \\
\frac{\Gamma; v : B \vdash^{\cap} E : \text{bool}}{\Gamma \vdash^{\cap} \{v : B \mid E\}}
\end{array}
\quad
\begin{array}{c}
\text{WF-VAR} \\
\frac{}{\Gamma \vdash^{\cap} \alpha}
\end{array}$$

$$\begin{array}{c}
\text{WF-FUN} \\
\frac{\Gamma; x : \tau_x \vdash^{\cap} \tau}{\Gamma \vdash^{\cap} x : \tau_x \rightarrow \tau}
\end{array}
\quad
\begin{array}{c}
\text{WF-POLY} \\
\frac{\Gamma \vdash^{\cap} \sigma}{\Gamma \vdash^{\cap} \forall \alpha. \sigma}
\end{array}$$

$$\begin{array}{c}
\text{WF-INTERSECT} \\
\frac{\Gamma \vdash^{\cap} \tau_1 \quad \Gamma \vdash^{\cap} \tau_2}{\Gamma \vdash^{\cap} \tau_1 \cap \tau_2}
\end{array}$$

Figure 3: Typing rules for Liquid Intersection Types.

$\Gamma = x : \{v \geq 0\}$, consider:

$$\mathcal{D}'_1 : \frac{\text{CONST} \frac{}{\Gamma \vdash_{\mathbb{Q}} - : (y : \text{int} \rightarrow \{v = -y\})} \quad \frac{\text{VAR-B} \frac{\Gamma(x) = \{v \geq 0\}}{\Gamma \vdash_{\mathbb{Q}} x : \{v = x\}} \quad \frac{\text{Valid}(x \geq 0 \wedge v = x \Rightarrow \top)}{\Gamma \vdash^{\cap} \{v = x\} \prec \text{int}} \prec\text{-BASE}}{\Gamma \vdash_{\mathbb{Q}} -x : \{v = -x\}} \text{SUB}$$

and:

$$\mathcal{D}_1 : \frac{\frac{\mathcal{D}'_1 \quad \frac{\text{Valid}(x \geq 0 \wedge v = -x \Rightarrow v \leq 0)}{\Gamma \vdash^{\cap} \{v = -x\} \prec \{v \leq 0\}} \prec\text{-BASE}}{\Gamma \vdash_{\mathbb{Q}} -x : \{v \leq 0\}} \text{SUB}}{\vdash_{\mathbb{Q}} \lambda x. -x : (x : \{v \geq 0\} \rightarrow \{v \leq 0\})} \text{FUN}$$

We can also derive $\vdash_{\mathbb{Q}} \lambda x. -x : (x : \{v \leq 0\} \rightarrow \{v \geq 0\})$ (similarly to the previous derivation, with the corresponding $\leq$ and $\geq$ symbols changed). Naming that derivation $\mathcal{D}_2$, we finally have:

$$\frac{\mathcal{D}_1 \quad \mathcal{D}_2}{\vdash_{\mathbb{Q}} \lambda x. -x : (x : \{v \geq 0\} \rightarrow \{v \leq 0\}) \cap (x : \{v \leq 0\} \rightarrow \{v \geq 0\})} \text{INTERSECT}$$

We omit the well-formedness and well-founded sub-derivations, since they are trivially constructed and use *int* to denote the type $\{v : \text{int} \mid \top\}$, that is, the common type for integer values.

2.3 Subject reduction

In order to prove soundness properties for our system we follow the approach of [13, 15]. The decidable notion of implication checking employed by the subtyping rules is a problem when it comes to prove a substitution lemma. So, instead we prove subject reduction for a version of the system with undecidable subtyping and unrestricted expressions in refinement predicates. The typing judgment in this system will be denoted by $\Gamma \vdash^{\cap} M : \sigma$. Then, we show that any derivation in the decidable system has a counter-part in the undecidable one. These results can be formalized as:

Theorem 1 (Subject Reduction)

- (Overapproximation) If $\Gamma \vdash_{\mathbb{Q}} M : \sigma$ then $\Gamma \vdash^{\cap} M : \sigma$;
- (Subject reduction) If $\Gamma \vdash^{\cap} M : \sigma$ and $M \rightsquigarrow N$ then $\Gamma \vdash^{\cap} N : \sigma$.

The detailed proofs can be found in [12].

Combining the two previous results guarantees that at run-time, for every well-typed term, taking an evaluation step preserves types.

3 Type inference

In this section we present our algorithm for inferring Liquid Intersection Types, figure 4. Before executing this algorithm we bind every sub expression using the *let-in* constructor. This transformation is closely related with A-Normal Forms [5] and is performed to force types of intermediate expressions to be pushed into the typing context. The algorithm we propose is built upon three main phases: **(i)** we use the ML inference engine to get appropriate types, serving as *type shapes* for Liquid Intersection Types; **(ii)** for some particular sub-terms a set of constraints is generated, ensuring the well-formedness of types and that subtyping relations hold, in order to infer sound types; **(iii)** taking qualifiers from $\mathbb{Q}$ we solve the generated constraints *on-the-fly*, much like as in classical inference algorithms.

$$\begin{aligned}
\text{Infer}(\Gamma, x, \mathbb{Q}) &= \text{if } \mathcal{W}(\text{Shape}(\Gamma), x) = B \text{ then } \{v : B \mid v = x\} \\
&\quad \text{else } \Gamma(x) \\
\text{Infer}(\Gamma, c, \mathbb{Q}) &= \text{ty}(c) \\
\text{Infer}(\Gamma, \lambda x.M, \mathbb{Q}) &= \text{let } (x : \hat{\tau}_1 \rightarrow \hat{\tau}'_1) \cap \dots \cap (x : \hat{\tau}_n \rightarrow \hat{\tau}'_n) = \text{Fresh}(\mathcal{W}(\text{Shape}(\Gamma), \lambda x.M), \mathbb{Q}) \text{ in} \\
&\quad \text{let } \tau''_i = \text{Infer}(\Gamma; x : \hat{\tau}_i, M, \mathbb{Q}) \text{ in} \\
&\quad \text{let } \mathcal{A} = \bigcap \left\{ (x : \hat{\tau}_j \rightarrow \hat{\tau}'_j) \mid \Gamma \vdash^\cap (x : \hat{\tau}_1 \rightarrow \hat{\tau}'_1) \cap \dots \cap (x : \hat{\tau}_n \rightarrow \hat{\tau}'_n) \right\} \text{ in} \\
&\quad \bigcap \left\{ (x : \hat{\tau}_k \rightarrow \hat{\tau}'_k) \mid x : \hat{\tau}_k \rightarrow \hat{\tau}'_k \in \mathcal{A}, \Gamma; x : \hat{\tau}_k \vdash_{\mathbb{Q}} \tau''_k \prec \hat{\tau}'_k \right\} \\
\text{Infer}(\Gamma, MN, \mathbb{Q}) &= \text{let } (x : \tau_1 \rightarrow \tau'_1) \cap \dots \cap (x : \tau_n \rightarrow \tau'_n) = \text{Infer}(\Gamma, M, \mathbb{Q}) \text{ in} \\
&\quad \text{let } \tau = \text{Infer}(\Gamma, N, \mathbb{Q}) \text{ in} \\
&\quad \bigcap [N/x] \left\{ \tau'_i \mid \Gamma \vdash_{\mathbb{Q}} \tau \prec \tau_i \right\} \\
\text{Infer}(\Gamma, \text{let } x = M \text{ in } N, \mathbb{Q}) &= \text{let } \hat{\tau} = \text{Fresh}(\mathcal{W}(\text{Shape}(\Gamma), \text{let } x = M \text{ in } N), \mathbb{Q}) \text{ in} \\
&\quad \text{let } \tau_1 = \text{Infer}(\Gamma, M, \mathbb{Q}) \text{ in} \\
&\quad \text{let } \tau_2 = \text{Infer}(\Gamma; x : \tau_1, N, \mathbb{Q}) \text{ in} \\
&\quad \text{let } \mathcal{A} = \bigcap \{ \hat{\tau}'_i \mid \Gamma \vdash^\cap \hat{\tau} \} \text{ in} \\
&\quad \bigcap \left\{ \hat{\tau}'' \mid \hat{\tau}'' \in \mathcal{A}, \Gamma; x : \tau_1 \vdash_{\mathbb{Q}} \tau_2 \prec \hat{\tau}'' \right\} \\
\text{Infer}(\Gamma, [\Lambda \alpha]M, \mathbb{Q}) &= \text{let } \sigma = \text{Infer}(\Gamma, M, \mathbb{Q}) \text{ in} \\
&\quad \forall \alpha. \sigma \\
\text{Infer}(\Gamma, [T]M, \mathbb{Q}) &= \text{let } \tau' = \text{Fresh}(T, \mathbb{Q}) \text{ in} \\
&\quad \text{let } \forall \alpha. \sigma = \text{Infer}(\Gamma, M, \mathbb{Q}) \text{ in} \\
&\quad \text{let } \mathcal{A} = \bigcap \{ \tau'_i \mid \Gamma \vdash^\cap \tau' \} \text{ in} \\
&\quad \sigma[\mathcal{A} / \alpha]
\end{aligned}$$

Figure 4: Type inference algorithm

3.1 Using Damas-Milner type inference

One key aspect of our inference algorithm is the use of the inference algorithm $\mathcal{W}$ [4] to infer ML types. Given the fact that a Liquid Intersection Type for a term is a refinement and intersections of the corresponding ML type, the types inferred by $\mathcal{W}$ act as *shapes* for our Liquid Intersection Types. Indeed, the function $\text{Shape}(\cdot)$ (figuring in the typing rules and in the inference algorithm) maps a Liquid Intersection Type to its corresponding ML type. For example, $\text{Shape}((x : \{v = 0\} \rightarrow \{v = 0\}) \cap (x : \{v \geq 0\} \rightarrow \{v \geq 0\})) = \text{int} \rightarrow \text{int}$.

In the inference algorithm, whenever $\mathcal{W}$ is called, we need to feed it with an environment containing exclusively ML types. This is done by lifting $\text{Shape}(\cdot)$ to environments, $\text{Shape}(\Gamma)$, by applying it to every binding in Γ .

The function $\text{Fresh}(\cdot, \cdot)$ takes an ML type and the set $\mathbb{Q}$ as input and generates a new Liquid Intersection Type that contains all the combinations of refinement expressions from $\mathbb{Q}$. Taking for instance the ML type $T = x : \text{int} \rightarrow \text{int}$ (we assume we can annotate types with the corresponding abstraction variable, so it is easier to use with refinements) and $\mathbb{Q} = \{v \geq 0, v \leq 0\}$, $\text{Fresh}(T, \mathbb{Q})$ would generate the Liquid Intersection Type

$$\begin{aligned}
&(x : \{v \geq 0\} \rightarrow \{v \geq 0\}) \cap \\
&(x : \{v \geq 0\} \rightarrow \{v \leq 0\}) \cap \\
&(x : \{v \leq 0\} \rightarrow \{v \geq 0\}) \cap \\
&(x : \{v \leq 0\} \rightarrow \{v \leq 0\})
\end{aligned}$$

3.2 Constraint generation

The constraints generated during inference serve as a means to ensure that the subtyping and well-formedness requirements are respected. In the presentation of the algorithm we borrow the notations from the typing rules, with $\Gamma \vdash^\cap \sigma$ standing for a well-formedness restriction over σ and $\Gamma \vdash^\cap \sigma \prec \sigma'$ constraining type σ to be a subtype of σ' .

The well-formedness constraints are generated for terms where a fresh Liquid Intersection Type is generated (λ -abstractions, let-bindings and type application). For a fresh generated Liquid Intersection Type, solving this kind of constraints will result in a type where the free variables of every refinement are in scope of the corresponding expression.

The second class of constraints are the subtyping ones, capturing relations between two Liquid Intersection Types. A constraint $\Gamma \vdash^\cap \sigma \prec \sigma'$ is valid if the type σ' is a super-type of σ , meaning that there is a type derivation using the subsumption rule to relate the two types.

The well-formedness and subtyping rules (Figure 3) can be used to simplify constraints prior to their solving. For instance, the constraint $\Gamma \vdash^\cap \tau_1 \cap \dots \cap \tau_n$ can be simplified to the set $\{\Gamma \vdash^\cap \tau_1, \dots, \Gamma \vdash^\cap \tau_n\}$. On the other hand, the constraint $\Gamma \vdash^\cap (x : \tau_1 \rightarrow \tau_2) \prec (x : \tau'_1 \rightarrow \tau'_2)$ can be further reduced to $\Gamma \vdash^\cap \tau'_1 \prec \tau_1$ and $\Gamma; x : \tau'_1 \vdash^\cap \tau_2 \prec \tau'_2$.

3.3 Constraint solving

We now describe the process of solving the collected constraints throughout the inference algorithm. This process will reduce to two different validity tests: a well-formedness constraint will, ultimately, reduce to the constraint of the form $\Gamma \vdash^\cap \{v : B \mid E\}$ and so it will amount to check if the type `bool` can be derived for E under Γ ; for the subtyping case, the simplification of constraints will result in a series of restrictions of the form $\Gamma \vdash^\cap \{v : B \mid E_1\} \cap \dots \cap \{v : B \mid E_n\} \prec \{v : B \mid E'_1\} \cap \dots \cap \{v : B \mid E'_m\}$, leading to check if $\llbracket \Gamma \rrbracket \wedge \llbracket E_1 \rrbracket \wedge \dots \wedge \llbracket E_n \rrbracket \Rightarrow \llbracket E'_1 \rrbracket \wedge \dots \wedge \llbracket E'_m \rrbracket$ holds.

Whenever well-formedness constraints are generated, these are solved before the subtyping ones. This step ensures only well-formed types are involved in subtyping relations. Well-formedness constraints arise when a *fresh* Liquid Intersection Type is generated, since that is when refinement expressions are plugged into a type. Such fresh types will be of the form $\tau_1 \cap \dots \cap \tau_n$, so the solution for a constraint of the form $\Gamma \vdash^\cap \tau_1 \cap \dots \cap \tau_n$ is the type $\bigcap \{\tau_i\}$, the intersection of all τ_i (with $1 \leq i \leq n$) such that $\Gamma \vdash^\cap \tau_i$. We assign this solution to a *temporary* type, denoted by $\mathcal{A}$, which will be used during the solving of subtyping constraints.

The subtyping constraints will ensure that inferred types only present refinement expressions capturing the functional behavior of terms. These will be used with λ -abstractions, applications and let-bindings. Except for applications, subtyping constraints are preceded by the resolution of well-formedness restrictions, and so it is the case that subtyping relations will be checked using the temporary type $\mathcal{A}$.

For the case of λ -abstractions, after generating the fresh Liquid Intersection Type $(x : \hat{\tau}_1 \rightarrow \hat{\tau}'_1) \cap \dots \cap (x : \hat{\tau}_n \rightarrow \hat{\tau}'_n)$, a series of calls to `Infer` are triggered, which we present via the syntax $\text{let } \tau''_i = \text{Infer}(\Gamma; x : \hat{\tau}_i, M, \mathbb{Q})$, with $1 \leq i \leq n$. These calls differ only on the type $\hat{\tau}_i$ of x pushed into the environment, implying that different types for M can be inferred. After solving the well-formedness constraints, we must remove from type $\mathcal{A}$ the refinement expressions that would cause the type to be unsound. We use the notation $x : \tau_k \rightarrow \tau'_k \in \mathcal{A}$ to indicate that $\bigcap \{x : \tau_k \rightarrow \tau'_k\}$ should be a supertype of $\mathcal{A}$, in the sense that it can be obtained from $\mathcal{A}$ using exclusively the rule $[\prec\text{-ELIM}]$ (taking an analogy with set theory, $\bigcap \{x : \tau_k \rightarrow \tau'_k\}$ would be a sub set of the intersections of $\mathcal{A}$). Then, the inferred type will be $\bigcap \{x : \hat{\tau}_k \rightarrow \hat{\tau}'_k\}$, such that $x : \hat{\tau}_k \rightarrow \hat{\tau}'_k \in \mathcal{A}$ and the constraint $\Gamma; x : \hat{\tau}_k \vdash^\cap \tau''_k \prec \hat{\tau}'_k$ is valid, that is, the type inferred for M under the

environment $\Gamma; x : \hat{\tau}_k$ is a subtype of $\hat{\tau}'_k$. As an example, consider $\mathbb{Q} = \{v \geq 0, v \leq 0, y = 5\}$, the term $\lambda x. -x$ and $\Gamma = \emptyset$. The inference procedure will start by generating the type:

$$\begin{aligned} &(x : \{v \geq 0\} \rightarrow \{v \geq 0\}) \cap \\ &(x : \{v \geq 0\} \rightarrow \{v \leq 0\}) \cap \\ &(x : \{v \leq 0\} \rightarrow \{v \geq 0\}) \cap \\ &(x : \{v \leq 0\} \rightarrow \{v \leq 0\}) \cap \\ &(x : \{v \geq 0\} \rightarrow \{y = 5\}) \cap \\ &(x : \{v \leq 0\} \rightarrow \{y = 5\}) \cap \\ &(x : \{y = 5\} \rightarrow \{v \geq 0\}) \cap \\ &(x : \{y = 5\} \rightarrow \{v \leq 0\}) \cap \\ &(x : \{y = 5\} \rightarrow \{y = 5\}) \end{aligned}$$

Then, with well-formedness constraints, and since no variable y is in scope, we are left with:

$$\begin{aligned} &(x : \{v \geq 0\} \rightarrow \{v \geq 0\}) \cap \\ &(x : \{v \geq 0\} \rightarrow \{v \leq 0\}) \cap \\ &(x : \{v \leq 0\} \rightarrow \{v \geq 0\}) \cap \\ &(x : \{v \leq 0\} \rightarrow \{v \leq 0\}) \end{aligned}$$

Finally, because of subtyping relations, the inferred type will be:

$$\begin{aligned} &(x : \{v \geq 0\} \rightarrow \{v \leq 0\}) \cap \\ &(x : \{v \leq 0\} \rightarrow \{v \geq 0\}) \end{aligned}$$

For application and let-bindings, solving subtyping constraints works in a similar manner as for λ -abstractions. The type of an application is inferred similarly as in [6]: for the function M with type $x : \tau_1 \rightarrow \tau'_1 \cap \dots \cap \tau_n \rightarrow \tau'_n$ and the argument N with type τ , the type of MN is $\bigcap \{\tau'_i\}$, such that $1 \leq i \leq n$ and $\Gamma \vdash^\cap \tau \prec \tau_i$ is checked valid.

3.4 Properties of inference

We were able to prove that our inference algorithm is sound with respect to the typing rules. This property is formalized as follows:

Theorem 2 (Soundness)

If $\text{Infer}(\Gamma, M, \mathbb{Q}) = \sigma$ then $\Gamma \vdash^\cap_{\mathbb{Q}} M : \sigma$.

4 Conclusion and future work

We presented a new type system supporting functional descriptions, via refinement types, and offering the expressiveness of intersection types. This type system can be used to derive more precise types than in previous refinement type systems.

To design a decidable system we adopted a style closely related to Liquid Types: the refinement expressions presented in types are exclusively collected from $\mathbb{Q}$, a global set of logical qualifiers, and the subtyping is decidable. We also impose that the type of an expression must be the intersection of refinements to its ML type, intersecting only types of the same form.

We also proposed an inference algorithm for Liquid Intersection Types. This algorithm takes as input an environment Γ , a term M and the set of qualifiers $\mathbb{Q}$, producing a correspondent Liquid Intersection

Type. Our inference algorithm uses the $\mathcal{W}$ algorithm to infer the shape of a Liquid Intersection Type, which is the ML type for that term. To determine which refinement expressions can be plugged into a type, the algorithm produces a series of well-formedness and subtyping constraints, solving them immediately after their generation. We have been able to prove that our algorithm is sound with the respected to the conceived typing rules.

Current and future work includes the study of completeness of type inference for our system and to extend decidable intersection type systems (of finite ranks [7, 8]) with type refinement predicates.

References

- [1] H. P. Barendregt (1984): *The Lambda Calculus, its Syntax and Semantics, Revised second edition*. North-Holland.
- [2] Henk Barendregt, Mario Coppo & Mariangiola Dezani-Ciancaglini (1983): *A filter lambda model and the completeness of type assignment*. *The journal of symbolic logic* 48(4), pp. 931–940.
- [3] M. Coppo & M. Dezani-Ciancaglini (1980): *An extension of the basic functionality theory for the λ -calculus*. *Notre Dame Journal of Formal Logic* 21(4), pp. 685–693.
- [4] Luis Damas & Robin Milner (1982): *Principal Type-schemes for Functional Programs*. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '82*, ACM, pp. 207–212.
- [5] Cormac Flanagan, Amr Sabry, Bruce F. Duba & Matthias Felleisen (1993): *The Essence of Compiling with Continuations*. In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93*, ACM, pp. 237–247.
- [6] Tim Freeman & Frank Pfenning (1991): *Refinement Types for ML*. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91*, ACM, pp. 268–277.
- [7] Trevor Jim (1995): *Rank 2 type systems and recursive definitions*. Massachusetts Institute of Technology, Cambridge, MA.
- [8] A. J. Kfoury & J. B. Wells (1999): *Principality and Decidable Type Inference for Finite-rank Intersection Types*. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '99*, ACM, pp. 161–174.
- [9] Kenneth Knowles & Cormac Flanagan (2010): *Hybrid Type Checking*. *ACM Trans. Program. Lang. Syst.* 32(2), pp. 6:1–6:34.
- [10] Charles Gregory Nelson (1980): *Techniques for Program Verification*. Ph.D. thesis, Stanford, CA, USA. AAI8011683.
- [11] C.-H. Luke Ong & Takeshi Tsukada (2012): *Two-level Game Semantics, Intersection Types, and Recursion Schemes*. In: *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II, ICALP'12*, Springer-Verlag, pp. 325–336.
- [12] Mário Pereira, Sandra Alves & Mário Florido (2014): *Liquid Intersection Types*. Technical Report, DCC. <http://www.dcc.fc.up.pt/~mariopereira/dcc-2014-4.pdf>.
- [13] Patrick M. Rondon, Ming Kawaguchi & Ranjit Jhala (2008): *Liquid Types*. In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, ACM, pp. 159–169.
- [14] Robert E. Shostak (1984): *Deciding Combinations of Theories*. *J. ACM* 31(1), pp. 1–12.
- [15] Niki Vazou, Patrick M. Rondon & Ranjit Jhala (2013): *Abstract Refinement Types*. In: *Proceedings of the 22Nd European Conference on Programming Languages and Systems, ESOP'13*, Springer-Verlag, pp. 209–228.